+++ date = '2025-03-13' draft = false title = 'Práctica 1' +++

En esta página se hará un análisis de un programa sobre una biblioteca

memory_management.h

```
#ifndef MEMORY_MANAGEMENT_H
#define MEMORY_MANAGEMENT_H

#include <stdio.h>

// Define a macro to enable or disable memory management display
#ifndef MEMORY_MANAGEMENT_DISPLAY
#define MEMORY_MANAGEMENT_DISPLAY 0
#endif
```

Las instrucciones #ifndef, #define se usan para que no se escriba más de una vez el código, esto nos evita errores, para cerrar el #ifndef se debe poner al final un #endif para cerrarla. El #include <stdio.h> es para incluir la librería estándar de c para la entrada y salida de datos.

```
// Counters for memory usage
extern int heap_allocations;
extern int heap_deallocations;
extern int stack_allocations;
extern int stack_deallocations;
```

Estas variables que se pueden observar son para contar las operaciones que se están haciendo con la memoria.

```
#if MEMORY_MANAGEMENT_DISPLAY
void displayMemoryUsage();
void incrementHeapAllocations(void *pointer, size_t size);
void incrementHeapDeallocations(void *pointer);
void incrementStackAllocations();
void incrementStackDeallocations();
#else
#define displayMemoryUsage() ((void)0)
#define incrementHeapAllocations(pointer, size) ((void)0)
#define incrementHeapDeallocations(pointer) ((void)0)
#define incrementStackAllocations() ((void)0)
#define incrementStackDeallocations() ((void)0)
#endif
```

Por último se encuentra la llamada de funciones que se van a utilizar más tarde para manipular la memoria.

memory_management.c

Para este programa se hace uso de la librería memory_management.h vista previamente.

Este programa usa una estructura llamada MemoryRecord para almacenar información sobre los bloques de memoria y se usa una lista para tener todo el registro de manera ordenada.

```
typedef struct MemoryRecord {
    void *pointer;
    size_t size;
    struct MemoryRecord *next;
} MemoryRecord;

MemoryRecord *heap_memory_records = NULL;
```

En esta funcion se añade un nuevo registro a la lista cuando se agregue memoria al heap.

```
void addMemoryRecord(void *pointer, size_t size) {
    MemoryRecord *record = (MemoryRecord *)malloc(sizeof(MemoryRecord));
    record->pointer = pointer;
    record->size = size;
    record->next = heap_memory_records;
    heap_memory_records = record;
}
```

Esta funcion tiene como propósito el eliminar un registro cuando se libere la memoria del heap

```
void removeMemoryRecord(void *pointer) {
    MemoryRecord **current = &heap_memory_records;
    while (*current) {
        if ((*current)->pointer == pointer) {
            MemoryRecord *to_free = *current;
            *current = (*current)->next;
            free(to_free);
            return;
        }
        current = &(*current)->next;
    }
}
```

Esta funcion incrementa el valor de un contador que cuenta las asignaciones que tenemos en el heap

```
void incrementHeapAllocations(void *pointer, size_t size) {
    heap_allocations++;
    addMemoryRecord(pointer, size);
    #if MEMORY_MANAGEMENT_DISPLAY
    printf("Memoria asignada en el heap: Puntero=0x%p, Tamano=%zu bytes\n",
pointer, size);
```

```
    #endif
}
```

Esta funcion incrementa el valor de un contador que cuenta las elimincaiones en el heap y elimina el registro.

```
void incrementHeapDeallocations(void *pointer) {
    heap_deallocations++;
    removeMemoryRecord(pointer);
    #if MEMORY_MANAGEMENT_DISPLAY
    printf("Memoria liberada en el heap: Puntero=0x%p\n", pointer);
    #endif
}
```

Estas funciones aumentan los contadores de asginaciones y eliminaciones en el stack.

```
void incrementStackAllocations() {
    stack_allocations++;
}

void incrementStackDeallocations() {
    stack_deallocations++;
}
```

Esta funcion imprime en pantalla como se ha usado la memoria.

```
void displayMemoryUsage() {
    printf("\n");
    printf("-------------------------------------------------\n");
    printf("|                  Uso de Memoria               |\n");
    printf("-------------------------------------------------\n");
    printf("| Heap (Memoria Dinamica)                       |\n");
    printf("|   Asignaciones: %-28d |\n", heap_allocations);
    printf("|   Liberaciones: %-28d |\n", heap_deallocations);
    printf("|-----------------------------------------------|\n");
    printf("| Stack (Variables Locales)                     |\n");
    printf("|   Asignaciones: %-28d |\n", stack_allocations);
    printf("|   Liberaciones: %-28d |\n", stack_deallocations);
    printf("-------------------------------------------------\n");

    printf("-------------------------------------------------\n");
    printf("|            Detalles de Memoria Heap           |\n");
    printf("-------------------------------------------------\n");
    printf("| Puntero          | Tamano (bytes)             |\n");
    printf("-------------------------------------------------\n");
    MemoryRecord *current = heap_memory_records;
    while (current) {
        printf("| 0x%-14p | %-27zu |\n", current->pointer, current->size);
```

```
        current = current->next;
    }
    printf("-------------------------------------------------\n");
    printf("\n");
}
```

biblioteca.c

Esta funcion agrega un nuevo libro a la biblioteca, asigna la memoria y actualiza el contador de libros.

```
void addBook(book_t **library, int* count) {
    book_t *new_book = (book_t *)malloc(sizeof(book_t)); // Asignación en el heap
    incrementHeapAllocations(new_book, sizeof(book_t));   // Registra la
asignación
    // Lógica para ingresar datos del libro...
    new_book->next = *library; // Agrega el libro al inicio de la lista
    *library = new_book;
    (*count)++;
}
```

Esta funcion lo que hace es buscar un libro en la librería por su identificador.

```
book_t* findBookById(book_t *library, int bookID) {
    book_t *current = library;
    while (current) {
        if (current->id == bookID) return current; // Retorna el libro si lo
encuentra
        current = current->next;
    }
    return NULL; // Retorna NULL si no lo encuentra
}
```

Gracias a estas funciones podemos acceder a los libros que tenemos disponibles en el momento.

```
void displayBooksRecursive(book_t *library) {
    if (!library) return;
    printf("\nID libro: %d\nTitulo: %s\nAutor: %s\n...", library->id, library-
>title, library->author);
    displayBooksRecursive(library->next); // Llamada recursiva
}

void displayBooks(book_t *library) {
    if (!library) printf("\nNo hay libros disponibles.\n");
    else displayBooksRecursive(library); // Inicia la recursión
}
```

Esta funcion nos permite agregar un nuevo miembro a la lista de miembros.

```
void addMember(member_t **members, int *memberCount) {
    member_t *new_member = (member_t *)malloc(sizeof(member_t)); // Asignación en
el heap
    incrementHeapAllocations(new_member, sizeof(member_t));      // Registra la
asignación
    // Lógica para ingresar datos del miembro...
    new_member->next = *members; // Agrega el miembro al inicio de la lista
    *members = new_member;
    (*memberCount)++;
}
```

Esta funcion nos permite prestar un libro disponible a un miembro activo, se disminuye la cantidad disponible de ese libro y se actualiza la lista de libros prestados a este miembro.

```
void issueBook(book_t *library, member_t *members) {
    // Busca el libro y el miembro...
    if (bookFound && memberFound) {
        bookFound->quantity--; // Disminuye la cantidad del libro
        memberFound->issued_count++;
        memberFound->issued_books = realloc(memberFound->issued_books,
memberFound->issued_count * sizeof(int)); // Reasigna memoria
        incrementHeapAllocations(memberFound->issued_books, memberFound-
>issued_count * sizeof(int)); // Registra la reasignación
        memberFound->issued_books[memberFound->issued_count - 1] = bookID; //
Agrega el libro a la lista del miembro
    }
}
```

Esta funcion permite regresar un libro que ha sido prestado, aumenta la cantidad disponible de este libro y se actualiza la lista de libros prestados al miembro.

```
void returnBook(book_t *library, member_t *members) {
    // Busca el libro y el miembro...
    if (bookFound && memberFound) {
        for (int i = 0; i < memberFound->issued_count; i++) {
            if (memberFound->issued_books[i] == bookID) {
                // Elimina el libro de la lista del miembro
                memberFound->issued_count--;
                memberFound->issued_books = realloc(memberFound->issued_books,
memberFound->issued_count * sizeof(int)); // Reasigna memoria
                incrementHeapAllocations(memberFound->issued_books, memberFound-
>issued_count * sizeof(int)); // Registra la reasignación
                bookFound->quantity++; // Aumenta la cantidad del libro
                break;
            }
        }
```

```
    }
}
```

Estas funciones liberan la memoria asignada para la biblioteca y los miembros.

```
void freeLibrary(book_t *library) {
    book_t *current = library;
    while (current) {
        book_t *next = current->next;
        incrementHeapDeallocations(current); // Registra la liberación
        free(current); // Libera la memoria
        current = next;
    }
}

void freeMembers(member_t *members) {
    member_t *current = members;
    while (current) {
        member_t *next = current->next;
        incrementHeapDeallocations(current->issued_books); // Registra la
liberación
        free(current->issued_books); // Libera la memoria de los libros prestados
        incrementHeapDeallocations(current); // Registra la liberación
        free(current); // Libera la memoria del miembro
        current = next;
    }
}
```

Estas funciones guardan y cargan la biblioteca a un archivo

```
void saveLibraryToFile(book_t *library, const char *filename) {
    FILE *file = fopen(filename, "w");
    book_t *current = library;
    while (current) {
        fprintf(file, "%d\n%s\n%s\n%d\n%s\n%d\n", current->id, current->title,
current->author, current->publication_year, genreToString(current->genre),
current->quantity);
        current = current->next;
    }
    fclose(file);
}

void loadLibraryFromFile(book_t **library, int *bookCount, const char *filename) {
    FILE *file = fopen(filename, "r");
    while (!feof(file)) {
        book_t *new_book = (book_t *)malloc(sizeof(book_t)); // Asignación en el
heap
        incrementHeapAllocations(new_book, sizeof(book_t)); // Registra la
asignación
```

```
        // Lógica para cargar datos del libro...
        new_book->next = *library;
        *library = new_book;
        (*bookCount)++;
    }
    fclose(file);
}
```

La primer funcion muestra a los miembros, y la segunda busca a un miembro por su identificador.

```
void displayMembers(member_t *members, book_t *library) {
    member_t *current = members;
    while (current) {
        printf("\nID miembro: %d\nNombre: %s\n...", current->id, current->name);
        for (int i = 0; i < current->issued_count; i++) {
            book_t *book = findBookById(library, current->issued_books[i]);
            if (book) printf("  Libro ID: %d\n  Titulo: %s\n...", book->id, book-
>title);
        }
        current = current->next;
    }
}

void searchMember(member_t *members, book_t *library) {
    int memberID;
    printf("\nIngresa el ID del miembro: ");
    scanf("%d", &memberID);
    member_t *current = members;
    while (current) {
        if (current->id == memberID) {
            printf("\nID miembro: %d\nNombre: %s\n...", current->id, current-
>name);
            for (int i = 0; i < current->issued_count; i++) {
                book_t *book = findBookById(library, current->issued_books[i]);
                if (book) printf("  Libro ID: %d\n  Titulo: %s\n...", book->id,
book->title);
            }
            return;
        }
        current = current->next;
    }
    printf("\nMiembro no encontrado.\n");
}
```

Mi Respositorio

Practica1